Making Pure Object-Oriented Languages Practical[*]

Craig Chambers
David Ungar

Stanford University[**]

Abstract

In the past, object-oriented language designers and programmers have been forced to choose between pure message passing and performance. Last year, our SELF system achieved close to half the speed of optimized C but suffered from impractically long compile times. Two new optimization techniques, *deferred compilation of uncommon cases* and *non-backtracking splitting using path objects*, have improved compilation speed by more than an order of magnitude. SELF now compiles about as fast as an optimizing C compiler and runs at over half the speed of optimized C. This new level of performance may make pure object-oriented languages practical.

1 Introduction

In the past, object-oriented language designers and programmers have been forced to choose between purity and performance. In a pure object-oriented language, all computation, even low-level operations like variable accessing, arithmetic, and array indexing, is performed by sending messages to objects. Although a message send may cost only one indirection more than a procedure call, a message send may cost much more than an *inlined* procedure call. Unlike a statically-bound procedure call, a message send refers to no single target method, and so the compiler cannot simply expand its destination in-line. Pure object-oriented languages thus exhibit high call frequencies which interfere with good performance. For example, the fastest commercial implementation of a pure dynamically-typed object-oriented language, ParcPlace Smalltalk-80[*] [GR83], runs a set of small C-style benchmarks at only 10% the speed of optimized C; this implementation contains techniques developed by Deutsch and Schiffman [DS84] that are widely considered to be the state of the art in software techniques for building fast implementations of pure object-oriented languages.

Even statically-typed (but pure) object-oriented languages like Trellis/Owl [SCW85, SCB+86], Eiffel [Mey86, Mey88], and Emerald [BHJL86, Hut87] must overcome the overhead of dynamically-dispatched message passing.[**] Static typing allows the compiler to check that an object will understand every message sent to it and perhaps to use a somewhat faster dispatching mechanism to implement messages. However, because an instance of a subclass can always be substituted for an instance of a superclass, and because subclasses can provide alternate overriding method implementations, static type-checking cannot determine in general the single target method invoked by a message. Since static typing alone does not enable static binding and inlining of messages, it cannot significantly reduce the overhead of message passing.

Hybrid object-oriented languages such as C++ [Str86, ES90] and CLOS [BDG+88] short-circuit the overhead (and consequently the benefits) of passing messages by including statically-bound procedure calls and primitive, non-object-oriented data types for simple things like numbers, arrays, and cons cells. These data types are accessed via built-in operators or procedure calls that are automatically inlined by the compiler to achieve good performance. However, it is only within pure object-oriented languages that the benefits of

[*] This work has been generously supported by an IBM graduate student fellowship, an NSF Presidential Young Investigator award, and grants from Sun, IBM, Apple, Cray, Tandem, NCR, TI, and DEC.

[**] Authors' present addresses: Craig Chambers, Department of Computer Science and Engineering, Sieg Hall, FR-35, University of Washington, Seattle, WA 98195; David Ungar, Sun Laboratories, MS 29-116, 2550 Garcia Ave., Mountain View, CA 94043.

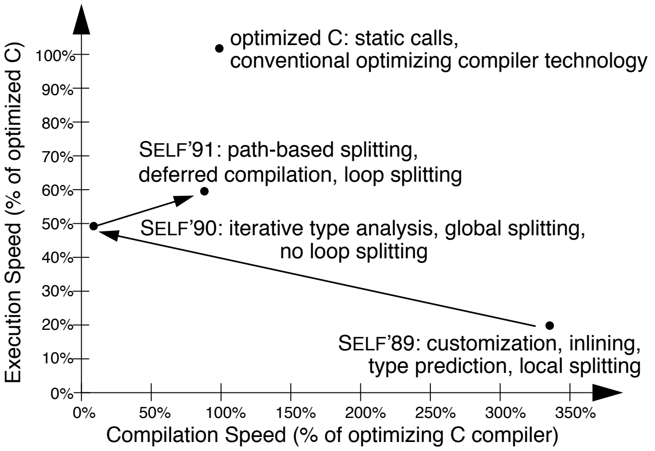
To appear in *OOPSLA'91 Conference Proceedings*, October, 1991.

[*] Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

[**] Even these "pure" languages restrict common built-in types like `integer` and `bool` to be non-object-oriented to get better performance.

object-oriented programming universally accrue for all code in a program. This fundamental trade-off between purity and performance has prevented the many programmers who need high performance from fully enjoying the benefits of object-oriented programming.

Over the last few years we have been working on bridging the gap between the performance of traditional languages and the performance of pure object-oriented languages. We are developing new implementation techniques and compiler optimizations for our implementation of SELF [US87], a pure dynamically-typed object-oriented language even harder to compile efficiently than Smalltalk-80. The following chart summarizes our progress to date, compared against optimized C (faster execution speed is higher on the graph, and faster compilation speed is farther to the right on the graph):



In 1989, we presented early results [CU89, CUL89] showing how our SELF system ran the same set of small C-style benchmarks at 20% the speed of optimized C, twice as fast as the ParcPlace Smalltalk-80 system. In 1990 we described more recent work [CU90] in which our SELF system ran the same benchmarks at 40% the speed of optimized C, another factor of two improvement over the early SELF system and a factor of four faster than ParcPlace Smalltalk-80. Unfortunately, the new techniques were slow: compiling a single benchmark took from tens to hundreds of seconds, and one technique, loop splitting, took an exponential amount of time and space and so could only be applied to the smaller benchmarks. Since our compiler runs dynamically at run-time to conserve compiled code space and minimize start-up time after a programming change, the sluggishness of the compiler sapped the overall performance of our system. Users could not tolerate minute-long pauses for compilation during interactive use.

This paper reports on work we have done this last year to re-engineer our techniques. New type analysis algorithms replace the expensive backtracking approach of the previous SELF compiler with path data structures to extract the same type information at a much lower cost. Additionally, the new system defers compilation of uncommon cases until they actually occur at run-time. As a result the compilation time of our optimizing SELF compiler has improved from a few minutes per benchmark to a few seconds, and we have been able to compile all our SELF code with full optimization. With this improvement, compile times for SELF now compete with those for optimized C. Gains in compilation speed are usually purchased at the expense of run-time performance, but the run-time performance of our new system has actually *improved*, to over 50% the speed of optimized C. Now that our optimization techniques have entered the realm of practicality, we hope that language designers and users can safely adopt a pure object-oriented model with message passing as the most basic mechanism for computation and rely on implementation techniques like ours to provide a level of performance, both run-time and compile-time, competitive with hybrid object-oriented languages and even traditional languages.

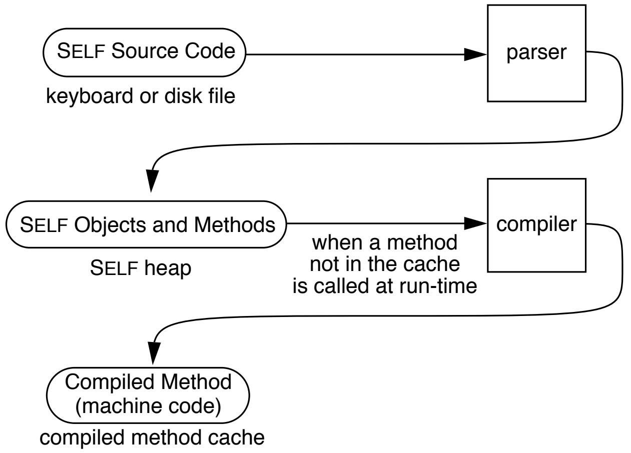
Section 2 presents an overview of the structure of the SELF compiler. Section 3 illustrates many of our new techniques using an excerpt from the `bubblesort` benchmark. Section 4 completes the discussion of our new techniques by extending the example to include a loop. Section 5 relates current run-time performance, compile-time performance, and compiled-code space costs for the current SELF compiler, the previous SELF compiler, the ParcPlace Smalltalk-80 system, the ORBIT optimizing compiler for T (a dialect of Scheme), and optimized C. Section 6 describes the status of our SELF implementation with hints of current and future work. Section 7 completes the paper with a brief discussion of related work.

2 Overall Structure of the SELF Compiler

Before explaining our new advances in type analysis and compilation, it is necessary to review the matrix in which they are embedded. Readers who are familiar with our past work may wish to skip this section.

2.1 Dynamic Compilation

The SELF system employs *dynamic compilation* to obtain better run-time performance than an interpreter while reducing compile-time and code-space costs over a conventional static compiler; dynamic compilation in SELF is similar to dynamic translation in the Deutsch-Schiffman Smalltalk-80 system [DS84]. When a programmer types in a program, a *parser* (corresponding to the Deutsch-Schiffman compiler) translates it into a simple byte-coded intermediate representation. Later, when the program is invoked, a *compiler* (corresponding to the Deutsch-Schiffman translator) compiles and optimizes the program, caching the resulting object code for future use.



2.2 Customization

Dynamic compilation offers new opportunities for efficient implementation not available in a traditional static compilation world. Although many classes* may inherit the same method, the SELF compiler compiles a separate, *customized* copy for each of them. In each copy, there is but a single class for `self`, and this knowledge allows the compiler to replace all dynamically-bound messages sent to `self` (such messages are a large fraction of the messages sent in SELF programs) with statically-bound procedure calls. Once a message has been statically-bound, more conventional techniques like inline substitution (*inlining*) can be used to reduce the overhead of the message even more. If the target method of the message is short (as is frequently the case in pure object-oriented languages), static binding and inlining can speed the message by an order of magnitude or more, especially if the inlined method can be optimized further in the context of the call.

2.3 Types in the SELF Compiler

Customization provides type information that enables compile-time message lookup for all sends to `self`. To perform similar optimizations for messages sent to other receivers, the SELF compiler performs *type analysis* to infer the exact class of message receivers. Like traditional data flow analysis, SELF's type analysis propagates the type binding information through the control flow graph. Unlike traditional data flow analysis, SELF's type analysis is interleaved with other techniques like message inlining and message splitting (described in section 3.3). These other techniques transform the control flow graph while the type analysis is labeling it. The need to label and transform the graph simultaneously empowers and complicates our techniques.

A type in the compiler specifies a non-empty set of values. A variable of a particular type at a particular point in the program is guaranteed to contain only values in the type's set of values at run-time at that point. The following table shows the different kinds of types used in the SELF compiler, chosen to support the optimizations.

name	**set description**	**static info**	**source**
constant	singleton set	compile-time constant	literals, constant slots, true and false type tests
integer subrange	set of sequential integer values	integer ranges	arithmetic and comparison primitives
class	set of all values with same class	format and inheritance	self, results of some primitives, integer type tests
unknown	set of all values	none	data slots, message results, up-level assignments
union	set union of types	one of several types	results of some primitives
difference	set difference of types	exclude certain types	failed type tests

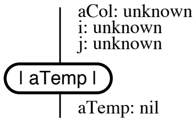
* Since SELF has no classes, our implementation introduces *maps* transparently to the user to provide similar information and space efficiency as classes [CUL89]. Thus in our system customization is based on the internal map of the receiver rather than its class. We will continue to use the class terminology in the rest of the paper for pedagogical reasons.

3 Type Analysis

To compute the static type information necessary for optimization, the compiler builds a mapping from variable names to types at each point in the program (i.e. between every node in the control flow graph). We will illustrate this process with a code fragment taken from the inner loop of the `bubblesort` benchmark that exchanges two elements in a vector (or any other collection that responds to `at:` and `at:Put:`):

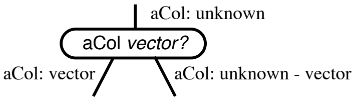
```
| aTemp |
. . .
aTemp: (aCol at: i).
aCol at: i Put: (aCol at: j).
aCol at: j Put: aTemp.
```

Each node in the control flow graph may alter the type bindings as type information propagates across the node. For example, a declaration of a local variable such as `aTemp` adds a new binding to the type mapping. Since local variables in SELF are always initialized to compile-time constants, each binding will initially be to some constant type. In our example, there is one local, `aTemp`, initialized to `nil`.

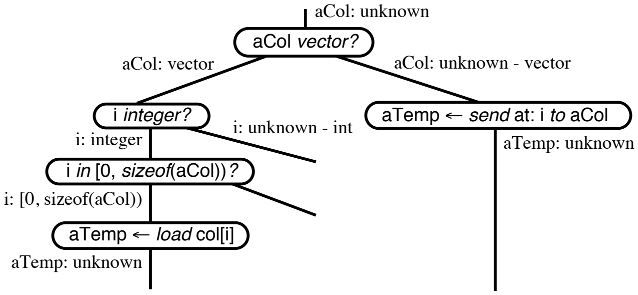


3.1 Type Prediction

Next the compiler must generate code for the first `at:` message. Since the receiver is `aCol`, and since the type of `aCol` is unknown, the compiler cannot statically determine the target method. However, the compiler predicts that there is a good enough chance that the receiver of `at:` is a vector to make it worthwhile to optimize this case. So the compiler inserts a type test for `aCol` before compiling the `at:` message.

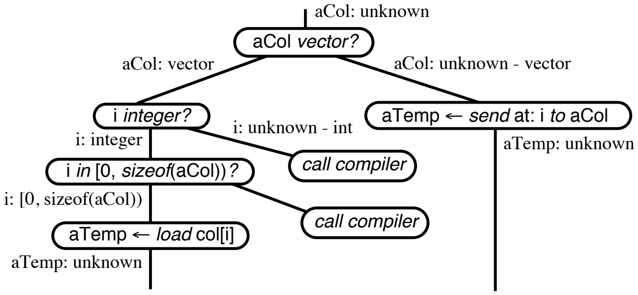


Now that two cases have been separated, the compiler is free to lookup and inline `at:` for the vector case along the left branch; it must still generate a full message send for the non-vector case. The inlined `at:` primitive includes a type check and a bounds check for the subscript argument, which result in two more branches.



3.2 Deferred Compilation

In the code generated for vectors, there are two branches that lead to primitive failure: one for a non-integer subscript argument and another for an out-of-bounds subscript argument. Just as the compiler is imbued with the knowledge that the receiver of `at:` is likely to be a vector, it is also informed that primitive failures are unlikely. In fact, they are so rare that it is not even worthwhile to spend time compiling code for them.* Accordingly, the SELF compiler defers compiling code for these uncommon cases, generating instead a stub that invokes the compiler. This optimization is new with the current SELF compiler.



Deferred compilation of uncommon cases dramatically reduces the time required for type analysis and conserves much compiled code space. In addition, deferred compilation of uncommon cases speeds and simplifies

* We would like to gratefully acknowledge John Maloney for pointing this out to us.

register allocation. In most traditional allocators (especially the faster ones), a variable is allocated to a single location for its entire lifetime. If the variable must survive across a call, then the allocator may require that the variable be allocated to a stack location for its entire lifetime rather than a register. In SELF code, many more calls remain in uncommon-case branches than in common-case branches, and so a variable cannot be allocated to a register only because of some calls in uncommon branches that the variable must survive. The performance of the common-case branch under such a register allocation strategy thus is adversely affected by the mere presence of uncommon branches, even if they are never executed. The previous SELF compiler attempted to solve this problem by allocating uses independently and allowing a variable to migrate to different locations during different portions of its lifetime, but this implementation was very slow. Deferred compilation of uncommon branches allows the new SELF compiler to use a simpler, faster register allocator.

In the rare case that the stub for an uncommon branch is executed, the compiler generates code for the uncommon branch in a separate compiled code object called an *uncommon branch extension*. This other routine reuses the stack frame created for the original common-case version and returns to the same place where the original common-case version would have returned to. When compiling such an uncommon branch extension, the compiler becomes very conservative. Since its predictions have been wrong for this method once already, it assumes it does not know much about the probabilities of the cases within the extension. All uncommon branches are fully generated in an uncommon branch extension (uncommon branch stubs are not used). This prevents recursive uncommon branch stub invocations which could lead to lengthy compile times. Also the compiler is biased in favor of saving compile time rather than generating better code when compiling an uncommon branch extension.

3.3 Message Splitting

The compiler has finally compiled the first `at:` message, splitting with respect to `aCol`'s type. It must now decide whether to merge the two control flow paths back together or to keep them apart. The two paths have different type information, and knowing this information may allow the compiler to optimize later messages (this is particularly true in this example in which there are three more messages sent to `aCol`). But keeping the two paths split apart takes up more compile time and compiled code space; this extra effort is wasted if the compiler won't end up making use of the information (such as if there were no more messages sent to `aCol`). Good heuristics for resolving this problem are central to achieving good run-time performance and good compile-time performance.

3.3.1 Eager Splitting: Never Merge

One extreme strategy would always keep branches apart and split everywhere possible (except at loop head merge nodes); we call this strategy *eager splitting*. Eager splitting has the advantages that it promises the best possible code quality and very simple forward-only type analysis. Unfortunately, the size of the control flow graph grows exponentially, and so pure forms of eager splitting are not practical. We have investigated several approaches to limiting the "eagerness" of this strategy, but with only limited success.

3.3.2 Reluctant Splitting and Paths: Merge Now, but Save the Information

An alternative strategy merges branches as soon as possible, but saves enough information to remove the merge later and split if need be. This *reluctant splitting* strategy allows the compiler to avoid unnecessary splitting.

The previous SELF compiler saved enough information using *merge types* (union types that implied that a merge node created the union) to decide whether or not to split the control flow graph, but could not split the merged type binding information into two more specific sets of type bindings. Hence, it had to reanalyze each of the split paths every time it split a merge to recalculate the split type information. Although such reanalysis occasionally revealed an opportunity for further optimization of a split path that was not possible in the original merged path, the backtracking nature of reanalysis made it very slow.

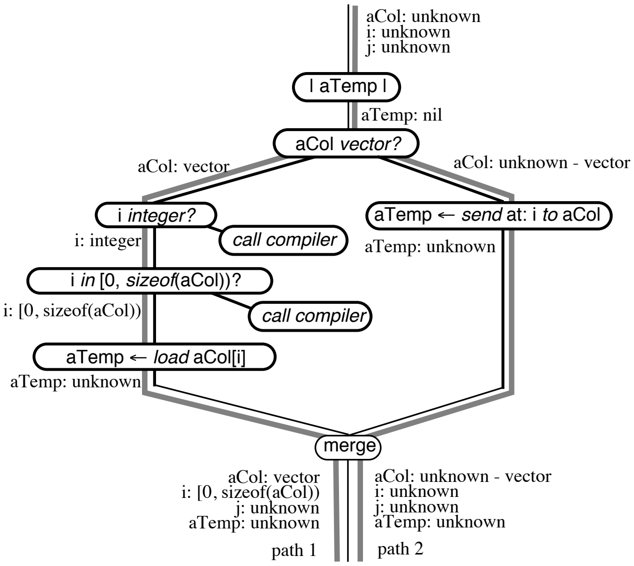
Our new type analysis algorithms implemented in the current SELF compiler do *not* reanalyze the parts of the control flow graph modified by splitting. As a result, the time to do type analysis becomes roughly linear with respect to the size of the resulting control flow graph. To split the type information as well as the control flow graph and avoid reanalysis, the new compiler uses a more detailed representation of type information based on *path objects*. Each path object represents a unique path through the control flow graph, and type information is conditional for particular path objects. At a branch node each path object splits into two separate paths, one per branch successor; at a merge node paths are not recombined (that would defeat the purpose of paths!) but instead are simply collected together to form a set of paths in the merge's successor node. A path object in reluctant splitting is analogous to a control flow branch in eager splitting, and consequently reluctant splitting using path objects has the potential for the same quality of type analysis as eager splitting, but at a fraction of the cost.

Once merge type information is replaced with per-path type information, splitting can be couched in terms of separating the subset of the possible execution paths that have a particular type binding (or combination of type bindings, or any other kind of information accumulated during type analysis, such as available expressions for common subexpression elimination) from those paths that do not. The new type information along the two split branches is easily calculated by simply filtering the type information by the appropriate set of path objects. Splitting is now much faster than the previous SELF compiler; copying the control flow graph nodes is relatively inexpensive, and paths relieve the compiler of the time to reanalyze the split control flow graph.

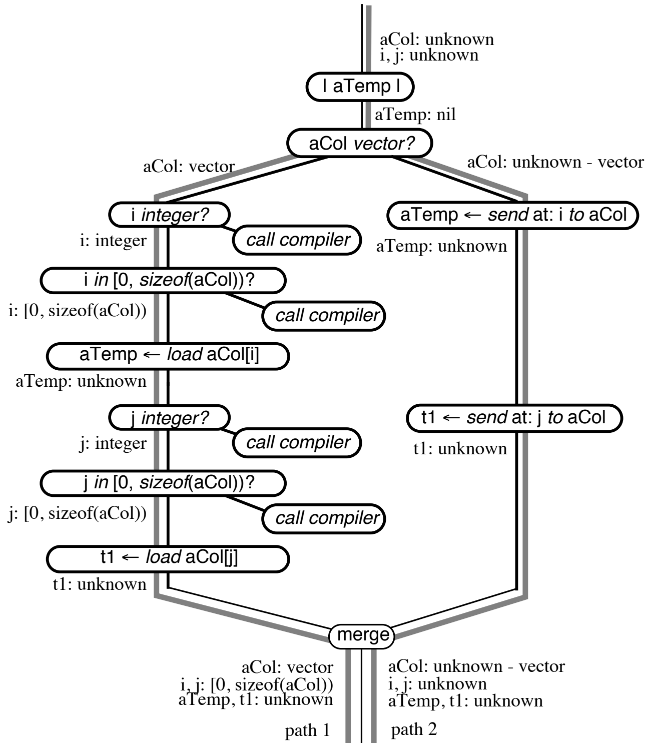
Of course, the path data structures have an associated cost in compiler complexity and compilation speed. At every branch node, the number of path objects doubles (one path along each outgoing branch for every single incoming path object), potentially leading to an exponential blow-up of paths that could swamp the compiler's type analysis. To combat this possibility, our compiler combines paths that have identical associated type information, since they will never be split apart. Fortunately, we have not observed exponential blow-up with this *global reluctant splitting* strategy for the SELF programs we have written so far.

To further reduce both overall compilation time and the risk of exponential blow-up, the SELF compiler normally uses a more conservative *local reluctant splitting* strategy. This approach forcibly combines all paths together into a single path after any control flow graph node that generates machine instructions, such as a message send node or a branch node but not an assignment node or a merge node. This has the effect of replacing each set of path-specific types with a single union type for the resulting combined path, thereby sacrificing some precision of type information to keep compilation fast. In the cases where this early path combining makes a difference, local reluctant splitting saves up to 30% of the compile time at a cost of up to 30% extra run time over global reluctant splitting [Cha91]. Perhaps surprisingly, global reluctant splitting sometimes enables optimizations which significantly shrink the size of the control flow graph and consequently occasionally compiles *faster* (and runs faster) than with local reluctant splitting.

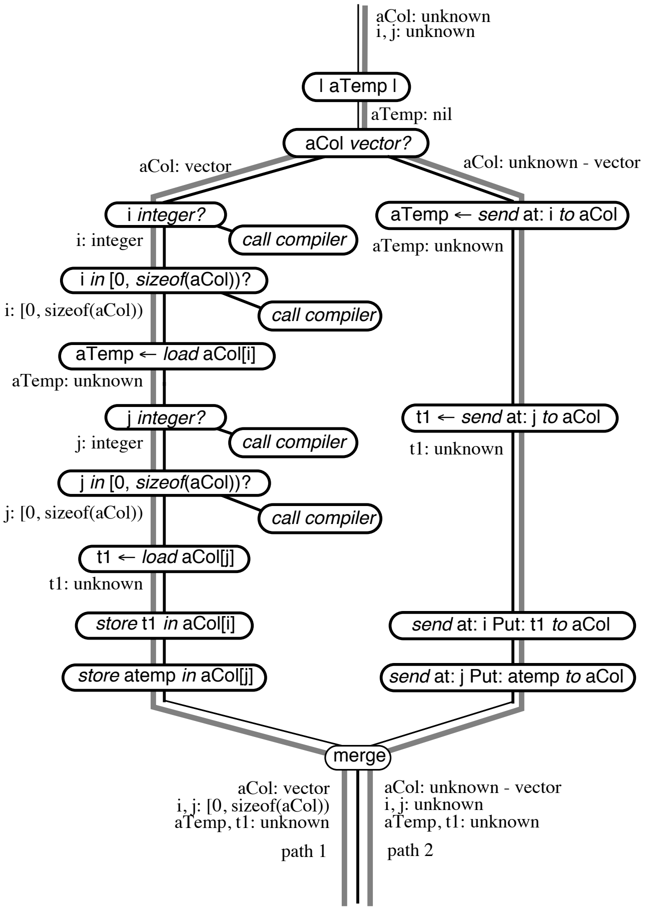
Here is our example up through the first merge, this time with the path information.



Now it is time to start using the paths. The compiler must generate code to send `at: j` to `aCol` and put the result in an anonymous temporary, called `t1` here. The type of `aCol` after the merge is either a vector (along path 1) or a non-vector (along path 2). Since the compiler keeps these two types separate using paths, it can do something more intelligent than inserting another type test to see if `aCol` is a vector. The compiler can use the path information to delay the premature merge until after the `at: j` message by *splitting* the merge and the `at: j` message into two copies, one along path 1 and one along path 2. In the path 1 case, `aCol` is known to be a vector, and the compiler can inline the `at: j` just as it did in the preceding `at: i` message. Along path 2, the compiler knows only that `aCol` is definitely not a vector, and so must fall back onto generating a full message send.



Now the compiler is faced with `aCol at: i Put: t1`. Again it splits path 1 from path 2, but this time the path objects uncover a hidden opportunity: the elimination of the type and bounds check of `i` along path 1. This extra type information about `i` is available because the compiler uses paths to recover the types of variables other than the one being split on. Paths support similar optimizations for the last message, `aCol at: j Put: aTemp`, as the next figure shows.



Every redundant type and bounds check has been eliminated, no effort has been wasted compiling uncommon cases, and none has been expended to reanalyze types after splits. Deferred compilation of uncommon branches and reluctant splitting based on paths enable these results.

4 Optimizing Loops

The previous example is taken from the inner loop of the `bubblesort` benchmark. Our techniques work especially well to optimize programs containing loops, frequently compiling *multiple versions* of a loop, each version optimized for different combinations of types.

Our type analysis technique for compiling loops is called *iterative type analysis*. The compiler first compiles the body of the loop assuming some type bindings at the loop head computed from the types at the loop entrance (the branch entering the loop from the top). It then checks to see whether the types computed at the loop tail are compatible with the loop head; if so it connects the loop tail to the loop head and is done with the loop. If the types are not compatible, the compiler attempts to split the loop tail to create a loop tail that is compatible with a loop head. If splitting won't help, then the analysis iterates, compiling a new loop body assuming more general types.

Compatibility of loop tails with loop heads must be defined carefully to preserve opportunities for optimization. Even though a loop head may have more general types than the types at a loop tail, and thus would be acceptable as a connecting loop head, the compiler avoids connecting a loop tail to any loop head that has less class type information. For example, if a loop head has some variable bound to the unknown type, while a loop tail has the same variable bound to the integer class type, the compiler will treat the loop head as incompatible; this ensures that the knowledge that the variable is an integer at the completion of the loop can be exploited in subsequent iterations. To avoid expending too much time and space on separate versions of loops, the compiler allows a loop tail to have a constant or subrange type while the loop head only has a class type (of the same class, of course); this sacrifices only a small amount of type information but saves a lot of compile time and compiled-code space.

Both the current SELF compiler and the previous SELF compiler follow this same plan when compiling loops. However, they differ in the details of how they answer the following questions:

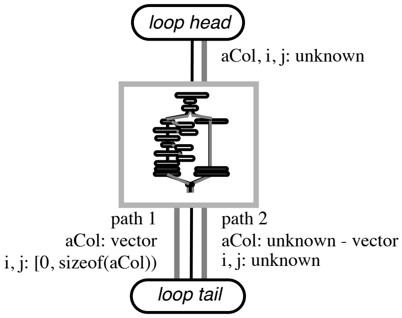
- *What should the initial type bindings be at the head of the loop?* The previous SELF compiler simply assumed the same types as the loop entrance. The current SELF compiler uses *early generalization* of constant and subrange types to the enclosing class types for any variables assigned within the loop. This usually saves an iteration over the simpler strategy used in the previous SELF compiler, although it occasionally sacrifices some type information.

- *What should the compiler do when it cannot connect a loop tail to any loop head?* The previous SELF compiler would replace the inadequate version of the loop body with a new fresh copy, with all split loop heads merged back together. It then allowed normal splitting to split the loop head back apart. This approach had the advantage that loop bodies should be quite compact, and since the previous SELF compiler didn't defer compilation of uncommon branches it ensured that all uncommon branches got merged together quickly to save compile time and compiled-code space. Unfortunately, the compiler ended up reproducing a lot of analysis to split the loop bodies apart over and over each iteration, and the backtracking type analysis was excruciatingly slow and space-consuming when reanalyzing split loop bodies. This compile-time performance problem hindered our ability to debug the compiler and its voracious appetite for compiler temporary space prevented us from compiling multiple versions of loops on any but the smallest benchmarks.

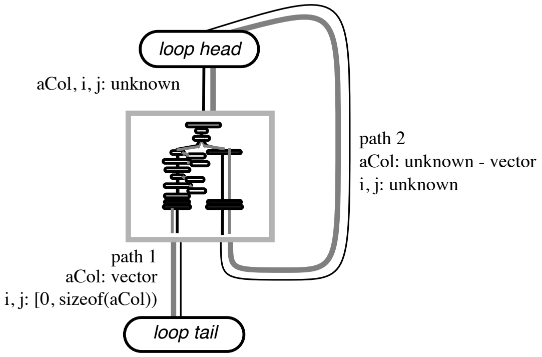
  The current SELF compiler avoids these pitfalls by adopting a simple, fast strategy. When a loop tail isn't compatible with any loop head, the compiler simply "unrolls" the loop for that one loop tail; the other versions of the loop remain unaffected. This strategy matches our forward-only type analysis scheme both in implementation simplicity and compilation speed. Its drawback is that in some situations this unrolling strategy uses more compiled code space, since there is no sharing of code among separate loop bodies. With deferred compilation of uncommon branches, this has not been a problem in practice.

We will illustrate these points by embedding our example within a simple endless loop; for the sake of simplicity we will ignore the loop counter testing and incrementing code that would be part of the real inner loop of the `bubblesort` benchmark.

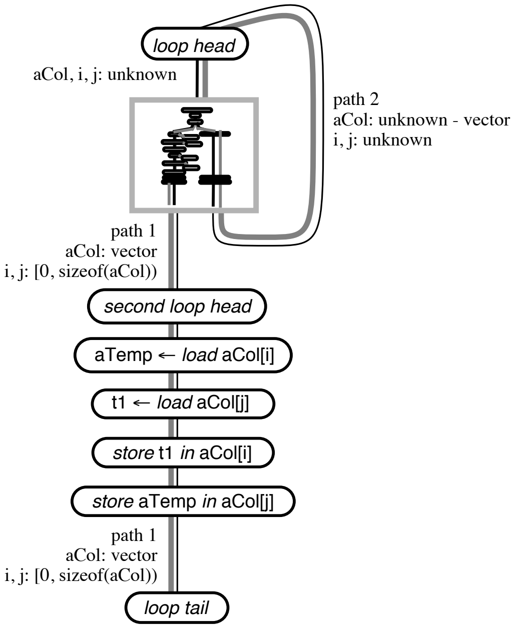
After compiling the loop for the first time, the control flow graph has one loop head, which could accept any types, and one loop tail with two paths.



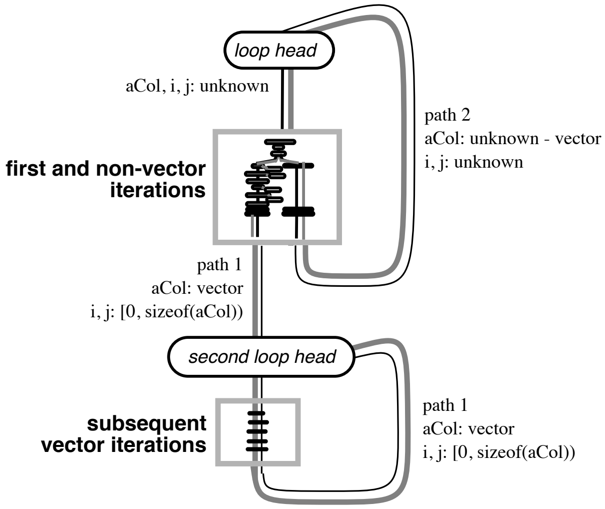
Although the compiler could choose to merely connect the tail up to the head, that would lose class type information on path 1, so the compiler splits path 1 off from path 2, and connects the loop tail for path 2 back to the head.



After connecting up the non-vector path, the compiler must generate code for the remaining path. Along this path the compiler already knows that `aCol` is a vector and that `i` and `j` are integer subranges.* This more specific type information enables a shorter, faster version of the loop to be generated with no type tests or bounds checks.



At this point, the types at the last loop tail are an exact match for the second loop head, and so the loop tail can be connected to the second loop head to finish our example.



At this point the compiler has ensured that most of the executions of the loop will be as good as can be. It has in effect compiled a separate version of the whole loop for the common case that `aCol` is a vector, hoisting the type test out of the subsequent iterations into the first iteration. All this has been accomplished without costly backtracking in the type analysis.

* In a more realistic example, `i` or `j` would be assigned within the loop and the compiler would generalize their types to the integer class type, so that the version of the loop to be compiled would handle more potential loop tails. In this pedagogical example, neither `i` nor `j` is assigned and so their types are left unchanged.

5 Performance Results

5.1 Methodology

In order to evaluate the performance impact of deferred compilation of uncommon cases and non-backtracking type analysis with loop unrolling and path objects, we compared the performance of three versions of SELF:

- SELF'90 is last year's SELF system [CU90], lacking both deferred compilation and non-backtracking type analysis. Partly because of lengthy compilation times, this system could not generate multiple copies of loops.
- SELF'91 (not deferred) is the current system, with non-backtracking type analysis, but with deferred compilation disabled. This system can be compared to SELF'90 to isolate the effect of non-backtracking type analysis.
- SELF'91 (normal) is the current system including both deferred compilation and non-backtracking type analysis. This system can be compared to SELF'91 (not deferred) to isolate the effect of deferred compilation. Both SELF'91 configurations used local reluctant splitting.

To measure execution and compilation times for a SELF benchmark, we first flushed the compiled code cache and then ran the benchmark 11 times. Consequently, the first run of the benchmark includes both compilation and execution, while the remaining 10 runs include only execution. Therefore, we calculate the execution time by taking the arithmetic mean of the times for the last 10 runs and calculate compilation time by subtracting the mean execution time from the time for the first run.

In order to evaluate the practicality of pure object-oriented languages, we standardized the processor, a lightly-loaded Sun-4/260 SPARC-based workstation, and measured several other systems:

- The standard Sun C compiler with optimization enabled (using the `-O2` option) established a goal for run-time performance. The graphs present performance relative to optimized C. (Appendix A contains the raw data.) Compilation time for optimized C includes the time to read and write files but not the time to link the resulting `.o` files together.
- In order to compare our approach to implementing a pure object-oriented language with competing approaches, we measured the ParcPlace Smalltalk-80 system (version 2.4) incorporating the Deutsch-Schiffman techniques [DS84].* As far as we know, this is the fastest implementation of any other system in which nearly every operation is performed by sending a dynamically-dispatched message. Variable accesses and some low-level control structures do not use messages in Smalltalk-80, unlike SELF.
- In order to compare our techniques against other systems supporting generic arithmetic, we also measured the ORBIT compiler (version 3.1) [KKR+86, Kra88] for T [RA82, Sla87], a dialect of Scheme [RC86]. ORBIT is well respected as a good optimizing compiler for a Scheme-like language. These data are labeled T/ORBIT (normal). Since nearly all benchmarks for Lisp-like systems measure programs that use integer-specific arithmetic, we also measure a version of the benchmarks using unsafe integer-specific arithmetic (e.g. `fx+` and `fx<` in T) and explicit indications to the compiler to inline certain functions (`define-integrable` in T). These data are labeled T/ORBIT (integer only). Compilation time includes the time to read and write files but not the time to load the generated file into the running T system.

In some ways, comparing these systems is like comparing apples to oranges. Our measured SELF system includes support for message passing at the most basic levels, user-defined control structures at the most basic levels, generic arithmetic, robust error-checking primitives, and support for source-level debugging. All these features are available in the SELF versions we measured. Neither the C version nor the T versions of the benchmarks use message passing or user-defined control structures, neither C nor the integer-specific version of T support generic arithmetic, neither C nor T compiled using ORBIT perform error checking for all primitives, and neither the optimizing C compiler nor T compiled using ORBIT support source-level debugging. We have directed much of our effort toward developing optimization techniques that coexist with the advantages of the SELF language and environment; programmers no longer need to choose between semantics and performance.

* Compile time and compiled code space measurements for Smalltalk-80 are unavailable.

We measured the eight Stanford integer benchmarks [Hen88] and the Richards operating system simulation benchmark [Deu88]. The C version of the `richards` benchmark is actually written in C++ version 1.2, translated into C using the standard `cfront` filter, and then optimized using the Sun C compiler. Only a few of the object-oriented features of C++ are used; for example there is only a single virtual function call (C++ terminology for a message send) in the entire benchmark. In the charts below we report the average results for seven small Stanford integer benchmarks and `puzzle` and `richards` separately; this separates the benchmarks into rough “equivalence classes” based on benchmark size. Raw data for each benchmark may be found in Appendix A.

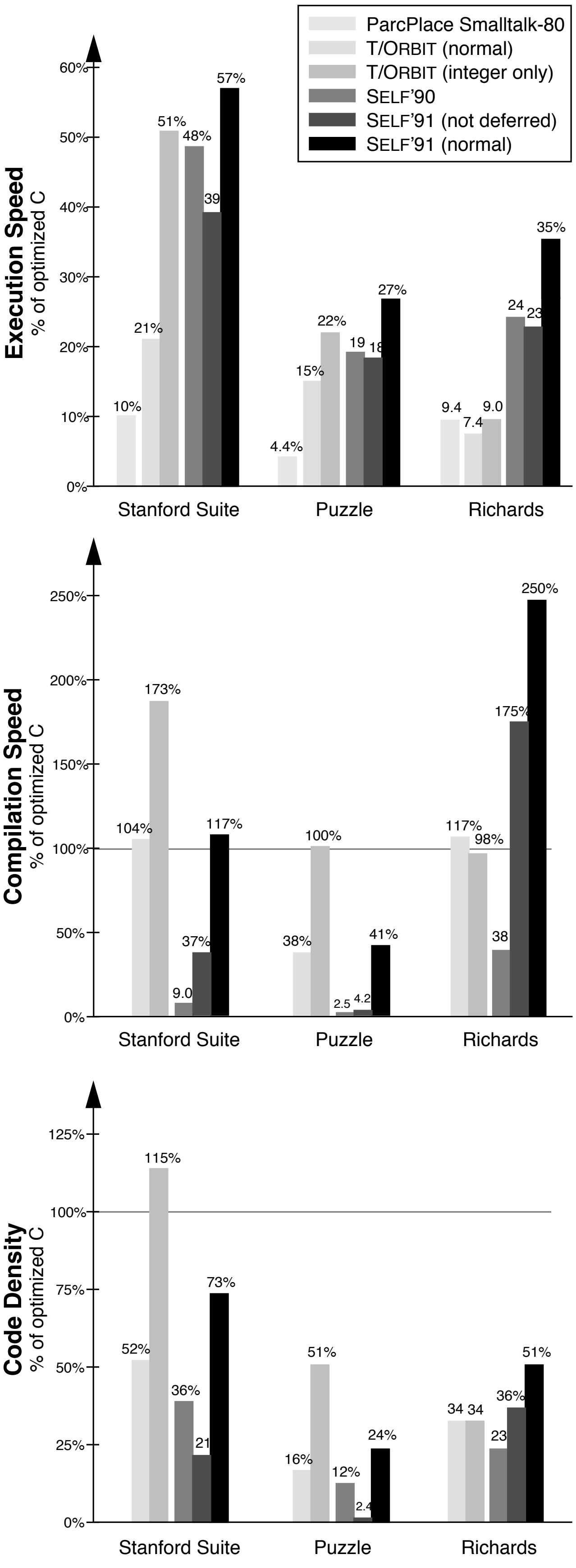
5.2 Results

The graphs to the right show the execution speed, compilation speed, and code density for our benchmarks, normalized to optimized C. Bigger bars are better.

5.2.1 Non-Backtracking Type Analysis and Deferred Compilation of Uncommon Cases

Comparing SELF’90 to SELF’91 (not deferred) reveals that eliminating backtracking speeds up compilation by a factor of two to four. Run-time performance and compiled-code space efficiency are roughly comparable for the two systems. Since other aspects of the SELF compiler and run-time system also changed between these two systems, it is difficult to make precise comparisons, but we can conclude that this technique succeeds at reducing compile times without significant penalties for the other metrics.

Comparing the two versions of SELF’91 with and without deferred compilation of uncommon cases reveals that this technique is an unqualified success, boosting compilation speed by up to a factor of 10, while simultaneously improving both execution performance and code space efficiency. In conjunction with non-backtracking type analysis, deferred compilation improves compilation speed by more than an order of magnitude over the previous SELF system.



5.2.2 Comparing SELF to Other Systems

The current SELF compiler runs the small Stanford integer benchmarks at well over half the speed of optimized C and more than five times faster than ParcPlace Smalltalk-80. The `puzzle` benchmark runs at over a quarter the speed of optimized C, again five times faster than Smalltalk-80. The `richards` benchmark runs at over a third the speed of optimized C, four times faster than Smalltalk-80; this level of performance is achieved partially because of recent work primarily by Urs Hölzle on extending the SELF system to speed polymorphic messages [HCU91].

These measurements suggest that our techniques would improve the speed of generic arithmetic even in non-object-oriented languages. Our SELF system runs between 50% and 250% faster than "normal" T programs compiled using the ORBIT compiler, and faster by 10% to 20% than even hand-tuned integer-specific T programs. Our results show that such hand-tuning and restricting of programs is no longer necessary to achieve good performance, and we would hope that future benchmarkers no longer resort to such violations of their languages in search of favorable performance comparisons.

Compilation speed for the current SELF system is comparable to optimized C and the normal version of T and ORBIT, is about half the speed of the restricted version of T and ORBIT, and is over twice the speed of C++ (recall that `richards` is written in C++). Part of SELF's compilation speed efficiency is because it does not read and write intermediate files but instead compiles native machine code directly into the same address space as the compiler. This avoidance of intermediate files and their accompanying overhead is one of the advantages of a dynamic compilation-based system.

SELF uses only a third more code space than optimized C for the small Stanford benchmarks and only twice as much code space for the `richards` benchmark. These results are better than the T/ORBIT combination when compiling normal T programs. We consider this amount of space usage to be reasonable, considering the relatively low cost of memory in today's workstations and the greater functionality provided by the SELF system over C and even T. We hope that these measurements allay any fears that our techniques require unreasonable amounts of extra space to be effective.

6 Implementation Status

The techniques described in this paper primarily were implemented in the SELF system in the summer and fall of 1990. We have been freely distributing this system, known as SELF Release 1.1, since January 1991. Over 150 sites around the world have a copy of our SELF implementation, and several medium-sized projects in SELF have been pursued by people outside our group.

Two compilers are distributed with this system: the latest SELF compiler described in this paper and the original SELF compiler described in [CU89] and [CUL89], which is less optimizing than the latest SELF compiler but compiles faster. When using the previous SELF compiler described in [CU90], a user would have to take a coffee break during a compilation, and many programs would fail to compile. With paths, the elimination of backtracking in the compiler's type analysis, and deferral of the compilation of uncommon cases, compile times for the latest compiler are merely distracting. To minimize this distraction, we are investigating adaptive recompilation strategies that reserve optimization for heavily-used methods [HCU91]; a crude form of this which first compiles methods using the original SELF compiler and later recompiles often-used methods using the latest SELF compiler is the standard configuration of our current system.

7 Previous Work

Other systems perform type inference over programs without explicit type declarations. ML [MTH90] is a statically-typed function-oriented language in which the compiler is able to infer the types of all procedures and expressions and do static type checking with virtually no type declarations. Researchers have attempted to extend type inference to object-oriented languages, with some success [Wan87, Wan88, Wan89, OB89, Rou90]. These approaches use type systems that describe an object's interface or protocol, not the object's representation or implementation. This abstract view of an object's type is best for flexible polymorphic type-checking, but provides little information for an optimizing compiler to speed programs. Our type analysis is more akin to traditional data flow analysis than type inference, in that it computes precise, time-varying, representation-level types for objects suitable for optimizations.

A different approach is taken by the Typed Smalltalk project [Joh86, JGZ88]. Users must annotate programs with type declarations for instance variables, class variables, global variables, and primitives, and then either run an inferencer to compute the types of methods and local variables [Gra89, GJ90] (which, like type inference in ML, provides the compiler with little information to support optimizations) or hand-declare selected methods and local variables with more specific representation-level type information. This representation-level type information is used by the TS optimizing compiler to perform run-time type casing and message inlining.

Published performance results for TS indicate that small Typed Smalltalk programs with explicit user-supplied type declarations run between five and ten times faster than Smalltalk programs executed by the Tektronix Smalltalk interpreter on a Tektronix 4405 68020-based workstation. Rough calculations based on the speed of the Deutsch-Schiffman Smalltalk implementation on a similar machine indicate that the TS optimizing compiler runs Typed Smalltalk programs about twice as fast as the Deutsch-Schiffman system runs comparable untyped Smalltalk-80 programs. This published result is still somewhere between two and three times slower than that achieved by our current SELF compiler, even though users do not add any declarations to SELF programs.

Recent unpublished performance results for very small benchmarks (smaller than any of the benchmarks we report in this paper) indicate that the current speed of Typed Smalltalk is close to the speed of our current SELF system, but unfortunately the current Typed Smalltalk system does not support generic arithmetic (integer arithmetic primitives do not check for overflow) [McC90]. Much of our work has been directed towards supporting both good performance and the complete language semantics; we found generic arithmetic particularly difficult to support efficiently in our SELF system and developed deferred compilation of uncommon branches and path-based splitting partially in response to this challenge.

8 Conclusions

The current version of the SELF compiler is both *effective* and *usable*. It executes small benchmarks at well over half the speed of optimized C, five times faster than the fastest existing implementation of any other pure object-oriented language with similar features, and with a compilation speed that is currently comparable to the optimizing C compiler. This new-found level of performance, both run-time and compile-time, hopefully will convince other language designers and language users that pure object-oriented languages are now practical.

The key technical contributions over our previous work on the SELF compiler are careful design and implementation of splitting strategies that rely only on path-based non-backtracking type analysis and deferred compilation of uncommon branches. Non-backtracking type analysis leads to up to an order-of-magnitude improvement in compilation speed by avoiding time-consuming reanalysis of split branches. Path objects are critical to realizing this non-backtracking goal without degrading the quality of the type analysis. Iterative type analysis is sped up further by eagerly generalizing the types of assignable local variables before loop analysis and by simply unrolling new copies of loops for loop tails that don't match any loop heads rather than starting the whole loop analysis over from scratch. Deferred compilation of uncommon branches exploits the skewed execution frequency distribution by only compiling those parts of the control flow graph that are likely to be executed. This technique increases compilation speed by nearly an order of magnitude and even improves execution speed by simplifying the type analysis and easing the register allocation problem. Deferred compilation fits in quite well with the on-demand dynamic compilation strategy used in our SELF implementation.

While the current SELF implementation is now usable and reliable, it still remains slower than desirable. We are pursuing techniques to reduce compiler pause times further, hopefully to the point at which SELF users forget that the compiler even exists.

Acknowledgments

We would like to express our heartfelt gratitude to the other members of the SELF group for their contributions and support: Urs Hölzle, Bay-Wei Chang, Ole Agesen, and Randall B. Smith.

References

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *OOPSLA'86 Conference Proceedings*, pp. 78-86, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. In *SIGPLAN Notices 23(Special Issue)*, September, 1988.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pp. 146-160, Portland, OR, June, 1989. Published as *SIGPLAN Notices 24(7)*, July, 1989.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA'89 Conference Proceedings*, pp. 49-70, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[CU90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 150-164, White Plains, NY, June, 1990. Published as *SIGPLAN Notices 25(6)*, June, 1990.

[Cha91] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. thesis, Stanford University, in preparation.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, pp. 297-302, Salt Lake City, UT, 1984.

[Deu88] L. Peter Deutsch. Richards benchmark source code. Personal communication, October, 1988.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Gra89] Justin Owen Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1989.

[GJ90] Justin O. Graver and Ralph E. Johnson. A Type System for Smalltalk. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pp. 136-150, San Francisco, CA, January, 1990.

[Hen88] John Hennessy. Stanford benchmark suite source code. Personal communication, June, 1988.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Programs using Polymorphic Inline Caches. In *ECOOP'91 Conference Proceedings*, Geneva, Switzerland, July, 1991.

[Hut87] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. Ph.D. thesis, University of Washington, 1987.

[Joh86] Ralph E. Johnson. Type-Checking Smalltalk. In *OOPSLA'86 Conference Proceedings*, pp. 315-321, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA'88 Conference Proceedings*, pp. 18-26, San Diego, CA, October, 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

[KKR+86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pp. 219-233, Palo Alto, CA, June, 1986. Published as *SIGPLAN Notices 21(7)*, July, 1986.

[Kra88] David Andrew Kranz. *ORBIT: An Optimizing Compiler for Scheme*. Ph.D. thesis, Yale University, 1988.

[McC90] Carl McConnell. TS performance data. Personal communication, October, 1990.

[Mey86] Bertrand Meyer. Genericity versus Inheritance. In *OOPSLA'86 Conference Proceedings*, pp. 391-405, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, New York, 1988.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[OB89] Atsushi Ohori and Peter Buneman. Static Type Inference for Parametric Classes. In *OOPSLA'89 Conference Proceedings*, pp. 445-456, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[RA82] Jonathan A. Rees and Norman I. Adams IV. T: a Dialect of Lisp or, LAMBDA: the Ultimate Software Tool. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 114-122, August, 1982.

[RC86] Jonathan Rees and William Clinger, editors. *Revised^3 Report on the Algorithmic Language Scheme*. In *SIGPLAN Notices 21(12)*, December, 1986.

[Rou90] Francois Rouaix. Safe Run-Time Overloading. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pp. 355-366, San Francisco, CA, January, 1990.

[SCW85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis Object-Based Environment, Language Reference Manual. Technical report DEC-TR-372, November, 1985.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA'86 Conference Proceedings*, pp. 9-16, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Sla87] Stephen Slade. *The T Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA'87 Conference Proceedings*, pp. 227-241, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987.

[Wan87] Mitchell Wand. Complete Type Inference for Simple Objects. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, pp. 37-44, Ithaca, NY, June, 1987.

[Wan88] Mitchell Wand. Corrigendum: Complete Type Inference for Simple Objects. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, p. 132, Edinburgh, Scotland, July, 1988.

[Wan89] Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pp. 92-97, 1989.

Appendix A Per-Benchmark Raw Data

The SELF'90 run time and compile time data are as reported in [CU90]. That system was not completely debugged and could not compile more than one copy of a loop. The system later was fixed and could compile more than one version of a loop, but was even slower to compile than the published system and still could not compile the larger benchmarks without running out of compiler temporary memory space. Compile times and code size data are not available for Smalltalk-80. The C version of `richards` is actually written in C++, preprocessed into C, then compiled with an optimizing C compiler; this partially accounts for its relatively slow compilation speed.

Run Times (ms)	**Smalltalk**	**T (normal)**	**T (int only)**	**SELF'90**	**SELF'91 (no defer)**	**SELF'91 (normal)**	**C (optimized)**
bubble	2700	1000	340	320	680	230	200
matrix multiply	4600	2500	900	700	640	600	280
perm	1400	1200	280	200	360	230	110
queens	860	640	240	260	250	180	92
quicksort	1300	1500	650	330	440	270	130
towers	1000	730	310	440	300	350	190
treesort	1100	1300	960	960	1000	930	870
puzzle	16000	4500	3100	3600	3900	2500.	690
richards	7700	9800	8100	3500	3200	2800.	730

Compile Times (s)	**T (normal)**	**T (int only)**	**SELF'90**	**SELF'91 (no defer)**	**SELF'91 (normal)**	**C (optimized)**
bubble	2.7	1.3	22	16	1.8	2.9
matrix multiply	3.0	1.5	30	9.8	3.8	2.9
perm	2.1	1.0	20	5.3	2.2	2.8
queens	3.4	1.6	25	8.2	4.6	3.1
quicksort	3.4	1.7	120	10	2.5	3.0
towers	3.4	3.5	7.6	1.9	1.3	3.7
treesort	3.5	2.4	7.0	6.5	2.1	3.9
puzzle	24	9.1	360	220	23	9.1
richards	12	14	36	7.7	5.4	13

Code Size (Kb)	**T (normal)**	**T (int only)**	**SELF'90**	**SELF'91 (no defer)**	**SELF'91 (normal)**	**C (optimized)**
bubble	4.7	1.9	5.9	21	2.2	2.7
matrix multiply	5.4	2.1	8.3	12	4.0	2.5
perm	3.6	1.3	7.1	7.9	2.9	2.4
queens	5.2	1.7	8.0	12	5.0	2.5
quicksort	5.8	2.7	10	20	3.6	2.8
towers	6.5	3.5	7.4	5.3	3.3	3.1
treesort	5.8	3.6	7.2	12	3.6	3.3
puzzle	32	9.9	41	210	21	5.0
richards	18	18	26	17	12	6.1